Simulação - funcionamento do Serviço de Urgência de uma Unidade Hospitalar

Raysa da Luz Oliveira

University of Beira Interior (UBI)
Covilhã, Castelo Branco, Portugal 6201-001
oliveira.raysa@gmail.com
http://di.ubi.pt

1 Introdução

O objetivo deste trabalho é implementar um modelo de simulação de um sistema relacionado com o funcionamento do Serviço de Urgência de uma Unidade Hospitalar. Este sistema é composto por quatro fases. A primeira é a fase da chegada do utente ao Serviço, onde o mesmo realiza a sua inscriçao. A segunda fase é a da triagem, onde nesta são definidas as prioriades de cada utente. A fase três é a fase de atendimento médico, onde o utente pode ser atendido por no mínimo um e no máximo dois médicos(ou duas especialidades). A última e quarta fase é a dos exames, onde nem todos os utentes pode ser direcionados à ela, nesta o utente que chegar pode fazer até dois exames. Este trabalho foi desenvolvido na linguagem de programação C, utilizando o Netbeans IDE. Nos próximos capítulos, segue a descrição com pormenores deste trabalho.

2 Implementação da Simulação Médica

2.1 Entidades da Simulação

As entidades desta simulação são estas:

1. Utente
2. Posto administrativo (inscrição)
3. Posto de Triagem
4. Médico
5. Funcionario Exame

2.2 Fluxograma

Como apresentado na introdução, o sistema é composto por quatro fases. Observando a figura 3, temos uma noção de como o sistema está implementado. Na fase de **Inscrição**, o cliente chega no sistema e entra em uma fila geral para realizar sua inscrição. Esta fase é composta por padrão, por dois servidores para atender a demanda de utentes. Na **triagem** o utente é questionado, para determinar a prioridade do tratamento com base na gravidade do seu estado. O

cliente pode seguir para uma determinada fila de acordo com a sua prioridade, fila 0 (vermelha) que corresponde ao atendimento imediato, fila 1 (laranja) se for considerado muito urgente (laranja), fila 2(amarelo), considerado urgente e fila 3(verde) pouco urgente. Os utentes serão atendidos respeitando esta distinção. Na fase de **atendimento médico**, o utente faz uma consulta, verifica se tem exames para fazer, se tiver exames ele irá fazer os **exames** e retorna para o mesmo médico que lhe receitou. Após finalizar com o primeiro médico, o utente pode ir ou não para outro médico. Se não for para outro médico, o utente sai do sistema. Caso ele faça outra consulta, é verificado se requer exames ou não, se requer, ele irá para a fase de exames e retorna novamente para o médico. Neste loop, temos duas verificações. Se a consulta for igual a 2, o utente sai do sistema. Da mesma forma, uma verificação para o exame, onde temos um vetor para guardar o retorno ao médico. Se ele estiver retornado duas vezes, foi porque foi duas vezes ao médico. Lembrando que o utente tem a possibilidade de não fazer exames, ou fazer um e no máximo fazer dois exames.

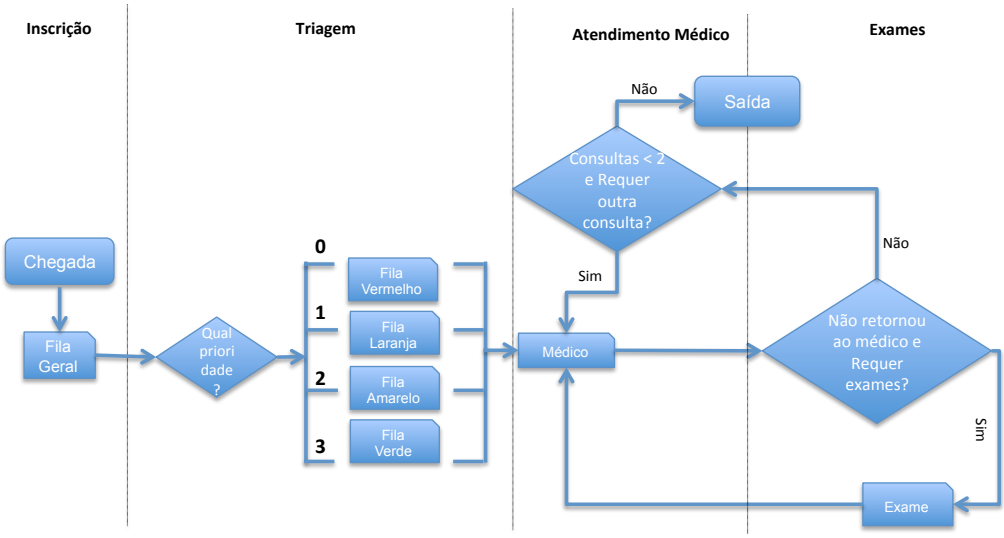

Figura 1: Fluxograma Simulação Médica

2.3 Código

Structs No arquivo tipos.h estão definidas todas as structs utilizadas no sistema. Temos a struct *status_fase* que tem esta estrutura:

```
1        struct status_fase{
2                int tempo_chegada;
3                int tempo_inicio_atendimento;
4                int duracao_atendimento;
5                int tempo_partida;
6        };
```

Esta struct é utilizada pelo utente para armazenar os tempos em cada fase do atendimento.

Há também uma struct para o utente, onde contém o id do utente, a struct status_fase do tipo status_fase onde tem acesso a todas as fases. Nesta estrutura, têm-se também a prioridade do utente, uma variavel para indicar e controlar os retornos aos medicos(retorno_medicos). O tamanho máximo deste vetor é a quantidade máxima de medicos que o utente pode consultar. Cada posição do vetor indica se o utente retornou ou não no médico de mesmo número do índice indicado. 0 indica que o utente nao retornou no medico da posicao indicada e um que já retornou. A variavél especialidades_medicas_consultadas, como o próprio nome já diz, indica os tipos de médico em que o utente foi atendido. O exames_medicos, que é um vetor que controla o total de exames solicitados por cada médico, o tamanho maximo dele é a quantidade máxima de de medicos que o utente pode consultar. Cada posição do vetor indica se o médico do índice indicado já prescreveu exames ao utente ou não. E a última variavél da estrutura é a total_atendimento_concluidos que indica o total de atendimentos médicos completos do utente.

```

        struct utente{
                int id;
                struct status_fase status_fase[TOTAL_FASES];
                int prioridade;
                int * retorno_medicos;
                int * especialidades_medicas_consultadas;
                int * exames_medicos;
                int total_atendimentos_concluidos;
        };
```

A estrutura node, onde contém o utente e o próximo.

```
typedef struct node{
        struct utente *utente;
        struct node * prox;
} node;
```

A estrutura fila, onde o node aponta pro inicio e fim da fila. Temos também a variavél quant_atual, onde através dela têm-se o controle da quantida atual da fila.

```
struct fila{
        node * inicio;
        node * fim;
        int quant_atual;
};
```

Têm-se também a estrutura fase, que contêm todos os dados para gerenciar os servidores e as filas de utentes de uma determinada fase.

```
struct fase{
        int numero_fase;
        int total_servidores;
        int total_filas;
        int tempo_max_atendimento;
        struct utente **servidores;
        struct fila **filas;
};
```

O numero_fase, começa em 0 (FASE1) e vai até TOTAL_FASES-1(FASE3). O total_servidores, que representa o total de servidores da fase (total de elementos do vetor servidores). O total_filas, que representa o total de filas da fase (total de elementos do veotr filas). Após ser chamado pelo servidor, o tempo_maximo_atendimento, é o tempo que o utente vai levar para ser atendido pelo servidor na fase. A variavél servidores é um vetor de ponteiro para utente, pois o que o servidor retorna é um utente. Cada posição do vetor representa o estado de um servidor. O valor NULL indica que o servidor está livre, caso contrário indica o utente sendo atendido. Usar utente **servidores é o mesmo que usar utente *servidores[], ou seja, a variável é um vetor de ponteiros para um utente. No entanto, se for usado [] é preciso indicar o tamanho do vetor. Usando * no lugar de [] estamos dizendo que o vetor não tem um tamanho pre-definido. struct fila **filas é o vetor de filas da fase. Se um determinado vetor tiver mais de uma fila, indica que cada fila é para utentes de uma determinada prioridade. Se tiver uma fila só, todos os utentes vão para tal fila.

A estrutra simulação apresenta estas variavéis:

```
1  typedef struct simulacao {
2          long seed;
3          int max_consulta_medicas_por_utente;
4          float intervalo_medio_entre_chegadas_utentes;
5          float probabilidade_de_utente_consultar_com_segundo_medico;
6          float probabilidades_prioridades[TOTAL_PRIORIDADES];
7          float probabilidades_especialidade_medica[
               TOTAL_ESPECIALIDADES_MEDICAS];
8          int total_minutos_simulacao;
9          int total_simulacoes;
10         struct fase fases[TOTAL_FASES];
11         struct fila *fila_utentes_finalizados;
12         int minuto_atual;
13         int imprimir_dados_utentes_individuais;
14 } simulacao;
```

A seed é a semente inicial para o gerador aleatorio. O número máximo de consultas por utente, é o maximo de vezes que o utente pode consultar. O intervalo médio entre as chegadas dos utentes, porque será utilizado na distribuição de poisson. A variavél probabilidade_de_utente_consultar_com_segundo_medico representa a possibilidade do utente consultar com o segundo médico ou não. A variavel probabilidades_prioridades[TOTAL_PRIORIDADES], onde cada valor define o intervalo de probabilidades para cada prioridade dos utentes do sistema (que vai de 0 a 3). Da mesma forma que a probalidade para as prioridades, temos as probabilidades para a especialidades médicas, onde cada valor define o intervalo de probabilidades para cada especialidade médica em que os utentes podem ser atendidos (que vai de 0 a 3). O total_minutos_simulação que é o total de minutos para executar cada simulação. o total_simulacoes que representa o total de simulações a serem executadas. A struct fases[TOTAL_FASES], onde tem todas as fases da simulação. A struct fila *fila_utentes_finalizados, que é uma fila para guardar todos os utentes que já finalizaram todos os atendimentos necessários para eles. Tal fila não faz parte de fato do fluxo da simulação, é usada apenas para calcular as estatísicas no final. O minuto_atual da simulação e por último a variavél imprimir_dados_utentes_individuais, onde indica se em cada

simulação serão impressos dados de cada utente a medida que eles passam pelas fases do sistema. 0 indica que não serão impressos e 1 que serão.

E por último, temos a struct estastiticas, que armazena os dados para uma simulação, como por exemplo, as médias.

```
struct estatisticas {
        int total_utentes_chegados;
        float media_total_utentes_chegados;
        float media_tempo_espera_fila_todas_fases;
        float media_tempo_espera_fila_por_fase[TOTAL_FASES];

        float media_duracao_atendimento_todas_fases;
        float media_duracao_atendimento_por_fase[TOTAL_FASES];

};
```

simulacao.c No arquivo simulacao.c temos a inicialização da seed, a inicialização do poisson. Lembrando que a distribuição de poisson é uma distribuição discreta de probabilidade aplicável a ocorrências de um evento em um intervalo especificado [1]. Esta distribuição foi utilizada neste trabalho para determinar as chegadas dos utentes no hospital.

Temos também a inicialização dos servidores, onde primeiramente é alocado um espaço para este vetor utilizando o malloc. Os servidores serão inicializado com null.

```
void inicializar_servidores_fase(struct fase *fase){
        fase->servidores = malloc(sizeof(struct utente *) * fase->
            total_servidores);
        for(int i=0; i<fase->total_servidores; i++){
                fase->servidores[i]=NULL;
        }
}
```

Neste arquivo têm-se também a inicialização das filas, onde o inicio e o fim da fila recebem null e a variavel quant_atual recebe 0.

A inicialização da simulaçao também é realizada aqui, setando todos os valores necessários para inicializar a simulação do sistema. Os valores passados para a inicialização são os valores passados por um arquivo de parametros.

A constatação de que o servidor está livre ou não, é feita pela função:

```
int servidor_esta_livre(struct utente *servidores[], int
    posicao_a_verificar);
```

Esta verifica se a posição indicada no servidor é igual a null ou não.

A função:

```
int gerar_prioridade(simulacao *sim)
```

É a função que gera de forma aleatória a prioridade de cada utente. São gerado valores de 0 à 1. É preciso ordenar as prioridades para que o código funcione, por exemplo:

```
[0.0 .. 0.1] = prioridade 0
]0.1 .. 0.3] = prioridade 3
]0.3 .. 0.6] = prioridade 1
]0.6 .. 1.0] = prioridade 2
```

As probabilidades neste caso, de 0 à 100%, são:
Prioridade 0: 10%, 1: 30%, : 2: 30%, 3: 30%. Temos que colocar os valores em limites.

A função:

```
int escolher_especialidade(simulacao *sim)
```

Esta função escolhe alguma especialidade para o utente consultar de forma aleatória. O valores para cada especialidade estão parametrizados no arquivo, tendo a porcetagem de 25% cada.

A função:

```
int vai_para_outro_medico(simulacao *sim, struct utente *utente){
        float aleatorio = rnd();

        if(utente->total_atendimentos_concluidos < sim->
            max_consulta_medicas_por_utente
                && aleatorio > sim->
                    probabilidade_de_utente_consultar_com_segundo_medico
                    ) {
                return 1;
        }

        return 0;
}
```

Verifica se o total de atendimento concluídos pelo utente é menor que o máximo de consultas médicas permitidas para o utente e ao mesmo tempo verifica aleatoriamente se o utente vai consultar com outro medico ou não. Se as duas condições forem verdade, ele vai para sua segunda consulta e retorna 1, caso contrário retorna 0.

A função:

```
int gerar_duracao_atendimento(struct fase *fase){
        int tempo_minimo_atendimento = 5;
        int aleatorio = rnd() * 100;
        int resto = (aleatorio % (fase->tempo_max_atendimento -
            tempo_minimo_atendimento));

        return resto + tempo_minimo_atendimento;
```

Retorna o tempo máximo que o utente vai ficar sendo atendido pelo servidor(atendente). Para cada fase há um tempo máximo de espera. Por exemplo, na primeira fase, a fase de incrição, não faz sentido o cliente ficar sendo atendido por 30 minutos. Então foi estipulado para tal fase um tempo menor, por exemplo, ele pode ficar sendo atendido nesta fase por no máximo 8 minutos. Entao o gerador irá gerar um valor aleatoriamente entre 0 e o tempo máximo. No caso desta função foi multiplicado o rnd() por 100, esse valor será convertido em um numero inteiro com ate 3 digitos. Como o meu tempo maximo de atendimento é definido pelo campo tempo_max_atendimento na fase, obtem-se o resto dividido por tal valor para garantir que o valor aleatório máximo será o definido em tal variável. Como o resultado do resto vai ser entre 0 e tempo_max_atendimento-1, mas deseja-se entre tempo_minimo_atendimento e tempo_max_atendimento, soma-se tempo_minimo_atendimento ao resto. Porém, inicialmente tinhamos valores entre 0 e tempo_max_atendimento-1, agora teremos entre 1 e tempo_max_atendimento+tempo_minimo_atendimento,

ou seja, o valor máximo será maior que o tempo máximo definido. Para evitar isso subtraiu-se o tempo_minimo_atendimento do tempo_max_atendimento no momento de calcular o resto.

A função:

```
int gerar_total_exames(simulacao *sim){}
```

Gera aleatoriamente um valor uniform entre 0 e o simulacao.max_consulta_medicas_por_utente para indicar o total de exames que o utente fará.

Destas 4 funções que seguem abaixo, a primeira verifica o total de utentes atualmente na fila em apenas uma fase, o total de utentes nas filas em todas as fases, o total de utente em atendimento em uma única fase e o total de utentes em atendimento em todas fases. Estas funções foram criadas para verificar se ainda tem utentes nas filas, se tiverem, mesmo que o tempo da simulação termine eles precisam terminar de ser atendidos. Ou seja, enquanto tiver utentes nas filas e em atendimentos, eles serão chamados para ser atendidos e finalizado os atendimentos.

```
int total_utentes_atualmente_em_fila(struct fase fase);

int total_utentes_atualmente_em_fila_em_todas_as_fases(struct fase fases
    [TOTAL_FASES]);

int total_utentes_em_atendimento(struct fase fase);

int total_utentes_em_atendimento_em_todas_fases(struct fase fases[
    TOTAL_FASES]);
```

As últimas funções do arquivo simulação.c são estas:

```
void liberar_filas_servidores_e_utentes_simulacao(simulacao *sim);
```

Que após a simulação, esta é chamada e as filas, o servidores e os utentes serão liberados.

E a função de imprimir os paramentros da simulação ,que estão no arquivo parametros-simulacao.txt.

```
void imprimir_parametros_simulacao(simulacao *sim);
```

fila.c Neste arquivo, fila.c, contém todas as funções referente ao manipulamento de uma fila. Estas 6 funções abaixo, faz as funções que toda implementação de fila faz, como inserir algo na fila, remover, limpar, verificar se está vazia ou não.

```
int vazia(struct fila *f);
struct fila * inicializar_fila();
void limpar_fila(struct fila *f);
void limpar_vetor_filas(struct fila * filas[], int tamanho_vetor_filas);
int inserir(struct utente * utente, struct fila *f);
struct utente *remover_inicio(struct fila *f);
void listar(struct fila *f);
```

Ja as funçoes abaixo, são funçoes mais específicas para o trabalho em questão. Neste mesmo arquivo, contém a função que imprime os dados dos utentes referente as fases, como por exemplo, tempo de chegada no hospital, tempo de atendimento, tempo de partida, etc.

Para calcular o tempo de espera na fila subtraimos o tempo_inicio_atendimento - tempo_chegada do utente. Para calcular o tempo de partida é somado o tempo_inicio_atendimento + duracao_atendimento.

```
1 void imprimir_utente(char mensagem[], struct utente * utente, int
      indice_fase);
2 int calcular_tempo_espera_na_fila_fase(struct status_fase status_fase);
3 int calcular_tempo_partida_do_utente_na_fila(struct status_fase
      status_fase);
```

A função que segue abaixo remove o primeiro utente que estiver na fila de espera, respeitando a prioridade.

```
1 struct utente *
      pesquisar_todas_as_filas_e_remover_utente_maior_prioridade(struct
      fase * fase);
```

A próximas que seguem abaixo inicializa qualquer vetor de inteiros com zero e a outra cria e inicializa os utentes.

```
1 void inicializar_vetor(int vetor[], int tamanho, int
      valor_para_inicializar);
2 struct utente * criar_e_inicializar_utente(int
      max_consulta_medicas_por_utente);
```

main.c O main.c é o arquivo principal desta simulação, onde de fato são simuladas as quatro fases do nosso sistema. No main é carregado o os parâmetros do arquivo. Se o arquivo não existir ou determinados parâmetros não forem setados, os valores para os parâmetros em falta serão os setados por default no main. A partir de tais parâmetros a simulação é inicializada. Primeiramente, é importante deixar claro que para gerar os números aleatórios randomicamente foi utilizado Mersenne twister [2]. A seed (um número usado para inicializar um gerador de números) utilizada neste gerador será a hora atual, assim, cada rodada gerará resultados diferentes. Coloque uma seed fixa e verá sempre os mesmos resultados. Na implementação foi multiplicado a seed por i para garantir que cada simulação terá uma seed diferente.

```
1 sim.seed = time(NULL)*(i+1);
```

Abaixo pode observar o laço da simulação, onde a simulação irá percorrer todos os minutos estabelecidos para a mesma. A simulação das quatro fases está dentro deste loop.

```
1   for(sim.minuto_atual = 1; sim.minuto_atual <= sim.
        total_minutos_simulacao; sim.minuto_atual++){
2           inserir_utente_na_fila_fase1(&sim);
3           chamar_utentes_em_espera_e_finalizar_atendimento_dos_utentes(&
                sim);
4   }
```

Para implementar a primeira fase do sistema foi criada a função inserir_utente_na_fila_fase1 que é especifica para esta fase. Temos a inserção apenas para fase 1, pois é apenas nesta fase que será gerada a chegada dos utentes no sistema.

```
1 void inserir_utente_na_fila_fase1(simulacao *sim){
2
3         if(rnd() < poisson()) {
```

```
4                   struct utente *utente = criar_e_inicializar_utente(sim->
                        total_utentes_chegaram, sim->
                        max_consulta_medicas_por_utente);
5
6           utente->prioridade = -1;
7
8           int indice_fila = 0;
9           if(sim->fases[0].total_filas > 1)
10                  indice_fila = utente->prioridade;
11          utente->status_fase[0].tempo_chegada=sim->minuto_atual;
12          utente->status_fase[0].tempo_inicio_atendimento = 0;
13          utente->status_fase[0].duracao_atendimento = 0;
14          utente->status_fase[0].tempo_partida = 0;
15          inserir(utente, sim->fases[0].filas[indice_fila]);
16          sim->total_utentes_chegaram++;
17          if(sim->imprimir_dados_utentes_individuais)
18                  imprimir_utente("inserido      ", utente, 0);
19          }
20  }
```

No primeiro if desta função é verificado se o resultado do poisson é menor que o rnd(). Se for menor indica que vai chegar um utente. Senão, nao chega utente no momento. Caso chegue um utente, ele será criado e inicializado. Nesta fase ainda não foi atribuído as prioridades para os utentes, pois as prioridades serão atribuidos apenas na fase de triagem. Com isto, utente->prioridade recebe -1 justamente para indicar que nenhuma prioridade foi definida nesta fase.

Se só existir uma fila para a fase, todos os utentes serão adicionados na fila 0, caso contrário, serão adicionados na fila de mesmo número de sua prioridade. Para isto é feito a verificação da quantidade de filas, se for maior que 1, a prioridade do utente é setada para a variavél indice_fila. Se a fila for igual a 1, o utente é inserido nesta.

Após esta função ser executada, a proxima função a ser chamada no main é a chamar_utentes_em_espera_e_finalizar_atendimento_dos_utentes. Como o próprio nome ja diz, esta função chama os utentes que estão nas filas de espera para o atendimento e finaliza os atendimentos dos utentes. Dentro desta função, existe outras duas que fazem isto de forma especifica.

```
1   void chamar_utentes_em_espera_e_finalizar_atendimento_dos_utentes(
        simulacao *sim){
2           for(int num_fase=0; num_fase < TOTAL_FASES; num_fase++){
3                   chamar_utente_para_atendimento_pelo_servidor(sim, &sim->
                        fases[num_fase]);
4                   finalizar_atendimento_utentes(sim, &sim->fases[num_fase
                        ]);
5           }
6   }
```

A função chamar_utente_para_atendimento_pelo_servidor procura uma posição livre no servidor. Se achar alguma posição livre, ou seja igual a null, o utente atendido será o de maior prioridade, se não tiver nenhum utente para ser atendido na fila de maior prioridade (FILA0), o proximo a ser atendido, será o da próxima prioridade(FILA1) e assim respectivamente.

Se algum cliente foi chamado da fila para iniciar atendimento pelo servidor, ou seja se utente diferente de null, o utente é atribuído ao servidor para indicar que o servidor agora está ocupado. Após isto é setado o tempo de inicio de atendimento e a duração do atendimento.

```
1  void chamar_utente_para_atendimento_pelo_servidor(simulacao *sim, struct
       fase *fase) {
2
3  for(int i=0; i < fase->total_servidores; i++){
4          if(servidor_esta_livre(fase->servidores, i)){
5                  struct utente* utente =
6                  pesquisar_todas_as_filas_e_remover_utente_maior_prioridade
                       (fase);
7                  if(utente!=NULL){
8                          fase->servidores[i] = utente;
9                  utente->status_fase[fase->numero_fase].
                       tempo_inicio_atendimento = sim->minuto_atual;
10                 utente->status_fase[fase->numero_fase].
                       duracao_atendimento = gerar_duracao_atendimento(fase
                       );
11                 if(sim->imprimir_dados_utentes_individuais)
12                         imprimir_utente("inicio atend.", utente, fase->
                               numero_fase);
13
14                 }
15         }
16 }
17 }
```

A proxima função a ser chamada é a:

```
1  void finalizar_atendimento_utentes(simulacao * sim, struct fase *
       fase_atual){
2
3  for(int i=0; i<fase_atual->total_servidores; i++){
4          struct utente * utente = fase_atual->servidores[i];
5
6          if(utente!=NULL){
7
8          struct status_fase status_fase_utente = utente->status_fase[
               fase_atual->numero_fase];
9          if(chegou_momento_de_finalizar_atendimento_utente(sim->
               minuto_atual, status_fase_utente)){
10                 int tempo_partida =
                       calcular_tempo_partida_do_utente_na_fila(
                       status_fase_utente);
11                 utente->status_fase[fase_atual->numero_fase].
                       tempo_partida=tempo_partida;
12                 fase_atual->servidores[i]=NULL;
13
14                 if(sim->imprimir_dados_utentes_individuais)
15                         imprimir_utente("finalizado    ", utente,
                               fase_atual->numero_fase);
16
17                 if(fase_atual->numero_fase <= 1) {
18                         struct fase *fase_seguinte = &sim->fases[
                               fase_atual->numero_fase+1];
19                         redirecionar_utente_para_fase_seguinte(sim,
                               fase_seguinte, utente);
20                 }
21                 else if(fase_atual->numero_fase==2)
22                         finalizar_atendimento_utente_fase_medico(sim,
                               fase_atual, utente);
23                 else if(fase_atual->numero_fase==3)
24                         finalizar_atendimento_utente_fase_exame(sim,
                               fase_atual, utente);
25                 }
26         }
27 }
28 }
```

Nesta função, o utente vai receber a posição do servidor da fase atual, depois disso, irá verificar se o servidor está atendendo algum utente (Se utente é dife-

rente de null). Se estiver, a função para verificar se chegou o momento de partir vai ser chamada, dentro dessa função é retornado verdade, caso o minuto atual seja maior ou igual que o tempo de partida. Se for verdade, significa q o atendimento do utente deve ser finalizado nesta fase e redirecionado para a proxima fase. E após isto, o servidor recebe null e fica livre novamente.

Nesta função temos algumas condições para comportamentos diferentes ao longo da simulação. Por exemplo, existe uma comparação para verificar se a fase atual for menor ou igual a 1. Se for, a fase seguinte nestes casos será sempre a fase_atual->numero_fase + 1.

Se a fase atual for igual a 2, vai ser chamada uma função especifica para esta fase. E se a fase atual for igual a 3, outra função distinta para tratamentos desta fase em particular será chamada.

Se a fase for igual a do médico(FASE2), será chamada esta função:

```
1  void finalizar_atendimento_utente_fase_medico(simulacao *sim, struct
       fase *fase_atual, struct utente *utente){
2          int idx_medico_atual = utente->total_atendimentos_concluidos;
3
4          if(utente->especialidades_medicas_consultadas[idx_medico_atual]
               == -1)
5          utente->especialidades_medicas_consultadas[idx_medico_atual] =
               escolher_especialidade(sim);
6
7          if(utente->retorno_medicos[idx_medico_atual]==1){
8                  utente->total_atendimentos_concluidos++;
9                  if(vai_para_outro_medico(sim, utente))
10                         redirecionar_utente_para_fase_seguinte(sim,
                               fase_atual, utente);
11                 else inserir_utente_fila_finalizados(sim, fase_atual,
                       utente);
12         }
13         else verificar_e_redirecionar_utente_fase_exame(sim, fase_atual,
                 utente);
14 }
```

O total_atendimentos_concluidos foi setado na inicialização da variavél como 0. Logo o índice do médico atual idx_medico_atual irá receber 0.

Observando o primeiro if desta função, é feita a verificação de que a especialidades_medicas_consultadas[idx_medico_atual] == -1. A variavél especialidades_medicas_consultadas é inicializada com -1, logo se ainda não foi definida a especialidade para qual o utente vai para a consulta atual, irá ser definida agora. De forma aleatoria está variavel será setada com uma especialidade.

O segundo if é uma condição para saber se o utente está retornando ao médico ou não. Se tiver retornando a variavél total_atendimentos_concluidos será incrementada. Após isto, temos outra condição para verificar se ele vai ou não para outro médico, se for ele será redirecionado para a mesma função. Lembrando que o utente só vai para outro médico se o total de atendimento concluídos for menor que max_consulta_medicas_por_utente e se probabilidade_de_utente_consultar_com_segundo_medico for verdade. Se não for para outro médico ele é inserido na fila dos finalizado, para computarmos as estatisticas da simulação.

Se o utente não estiver retornando para algum médico ele vai pra fase de verificação e redirecionamento para exame (verificar_e_redirecionar_utente_fase_exame).

Nesta próxima fase, utilizamos esta função:

Esta função gera de forma aleatória um total de exames para o utente, e a variavel exames_medicos[idx_medico_atual] irá receber este valor. Se o utente não tiver exames para fazer, ele sai do sistema e é guardado na fila de finalizados para computar estatísticas no final da simulaçao. Há um if com uma verificação para saber se o total de exames do utente é igual a 0. Se for, como o utente não fará exames, ele não retornan ao médico. Assim, seta o retorno para o médico atual como zero para indicar que ele já passou pelo médico mas náo retorna e a variavél total_atendimentos_concluidos é incrementada.

Se o utente for consultar com outro médico, ele será redirecionado para uma fila da fase do médico para que ele aguarde atendimento de outro médico. Se ele não for para outro médico, ele será inserido na fila dos utentes finalizados.

Se o total de exames gerado for maior que 0, ele irá fazer exames, ou seja ele será redirecionado para fase de exame. Nesta função ele apenas verifica se o utente tem exames ou não. Se não tiver ele pode sair do sistema ou consultar outro médico, caso ele tenha, este será redirecionado.

```
1  void verificar_e_redirecionar_utente_fase_exame(simulacao *sim, struct
       fase *fase_atual, struct utente *utente){
2          int idx_medico_atual = utente->total_atendimentos_concluidos;
3          int total_exames_utente = gerar_total_exames(sim);
4          utente->exames_medicos[idx_medico_atual] = total_exames_utente;
5
6          if(total_exames_utente==0){
7                  utente->retorno_medicos[idx_medico_atual] = 0;
8                  utente->total_atendimentos_concluidos++;
9
10                 if(vai_para_outro_medico(sim, utente))
11                         redirecionar_utente_para_fase_seguinte(sim,
                               fase_atual, utente);
12                 else inserir_utente_fila_finalizados(sim, fase_atual,
                       utente);
13         }else {
14                 struct fase *fase_seguinte = &sim->fases[fase_atual->
                       numero_fase+1];
15                 redirecionar_utente_para_fase_seguinte(sim,
                       fase_seguinte, utente);
16         }
17 }
```

Na fase 3, aonde os utentes fazem os exames, indica que o utente agora vai retornar ao médico após ter finalizado os exames. O utente nesta fase será redirecionado para a fase do atendimento médico. solicitados por ele*/

```
1  void finalizar_atendimento_utente_fase_exame(simulacao *sim, struct fase
       *fase_atual, struct utente *utente){
2          int idx_medico_atual = utente->total_atendimentos_concluidos;
3
4          utente->retorno_medicos[idx_medico_atual] = 1;
5          struct fase *fase_anterior = &sim->fases[fase_atual->numero_fase
               -1];
6          redirecionar_utente_para_fase_seguinte(sim, fase_anterior,
               utente);
7  }
```

Quando o utente retorna para a fase dos médicos (fase 2), ele vai com a variavél retorno_medicos[idx_medico_atual] = 1. Tendo este valor, após o retorno o utente entra nesta condiçao:

```
1  if(utente->retorno_medicos[idx_medico_atual]==1){
2          utente->total_atendimentos_concluidos++;
3          if(vai_para_outro_medico(sim, utente))
4                  redirecionar_utente_para_fase_seguinte(sim, fase_atual,
                       utente);
5          else inserir_utente_fila_finalizados(sim, fase_atual, utente);
6  }
```

Onde após a segunda consulta ele não poderá mais consultar com outro médico, desta forma ele será inserido na fila dos utentes finalizados.

A simulação será executada seguindo este fluxo. No arquivo parametros-simulacao.txt será definido o total de vezes que está simulação será executada.

estatisticas.c

No arquivo estatisticas.c,estão todas as funções necessárias para fazer os calculos estatistícos para a simulação. Desde função para calcular média de espera em cada fase e a média geral em todas as fases, calcular a média do tempo de atendimento dos utentes em cada fase e a média de atendimento geral, e contabilizar todos os utentes que chegaram no sistema e a média deste. Foi criado um vetor_estatisticas[] para guardar as estatisticas de cada simulação.

parametros-simulacao.txt Dentro do arquivo parametros-simulacao.txt, estão todos os valores que serão passados por paramentro. Caso deseja trocar algum valor de algum parametro, basta editar tal arquivo. Se por acaso algum dos parametros listados neste arquivo estiver vazio, ele utiliza os valores defaults que estão sendo inicializados no main.c .

Outputs do sistema Após o termino da simulação, será mostrado na tela as média de espera em cada fase e a média geral em todas as fases, a média do tempo de atendimento dos utentes em cada fase e a média de atendimento geral, e todos os utentes que chegaram no sistema e a média dos utentes chegados.

```
--------------------------------------------------------------------------------


ESTATÍSTICAS FINAIS PARA TODAS AS SIMULAÇÕES
        Média do tempo de espera na fila para fase 0: 0.21 minutos
        Média do tempo de espera na fila para fase 1: 0.77 minutos
        Média do tempo de espera na fila para fase 2: 6.53 minutos
        Média do tempo de espera na fila para fase 3: 0.30 minutos
        Média do tempo de espera na fila para todas a fases: 1.95 minutos

        Média de duração atendimento para fase 0: 5.99 minutos
        Média de duração atendimento para fase 1: 9.49 minutos
        Média de duração atendimento para fase 2: 16.93 minutos
        Média de duração atendimento para fase 3: 12.28 minutos
        Média de duração de atendimento para todas a fases: 11.17 minutos

        Média do total de utentes chegados no sistema: 33.21
        Total de utentes chegados: 6642

        Tempo da simulação por rodada: 420 minutos
        Tempo total de simulação: 84000 minutos
```

Figura 2: Resultados - Simulação

Acima está o resultado da simulação. Foram utilizado estes parametros para obter estes resultados:

```
total_minutos_simulacao = 420
total_simulacoes = 200
max_consulta_medicas_por_utente = 2
probabilidade_de_utente_consultar_com_segundo_medico = 0.5
total_servidores_fase0 = 2
total_servidores_fase1 = 2
total_servidores_fase2 = 4
total_servidores_fase3 = 3
total_filas_fase0 = 1
total_filas_fase1 = 4
total_filas_fase2 = 4
total_filas_fase3 = 4
tempo_max_atendimento_fase0 = 8
tempo_max_atendimento_fase1 = 15
tempo_max_atendimento_fase2 = 30
tempo_max_atendimento_fase3 = 30
probabilidades_prioridade0 = 0.1
probabilidades_prioridade1 = 0.3
probabilidades_prioridade2 = 0.6
probabilidades_prioridade3 = 1.0
probabilidades_especialidade_medica0 = 0.25
probabilidades_especialidade_medica1 = 0.50
probabilidades_especialidade_medica2 = 0.75
probabilidades_especialidade_medica3 = 1.0
imprimir_dados_utentes_individuais = 0
```

Figura 3: Parametros - Simulação

Este resultado é satisfatório, pois os utentes com esta configuração estão sendo atendidos de forma rápida, não obtendo um tempo de espera grande. O total de vezes que a simulação foi executada foi igual a 200, e o tempo de simulação para cada rodada foi o de 420 minutos (7 horas).

Referências

1. How to generate random timings for a poisson process. [Online]. Available: http://preshing.com/20111007/how-to-generate-random-timings-for-a-poisson-process/
2. G. Kuenning. Use pcg instead! [Online]. Available: https://www.cs.hmc.edu/~geoff/mtwist.html